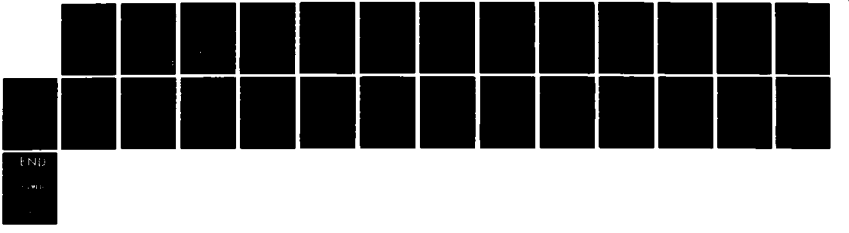
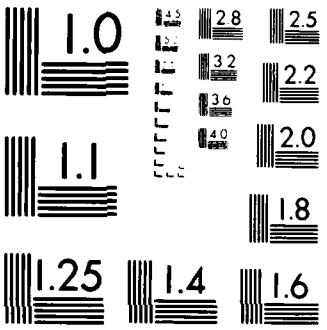
AD-A149 812 PARALLEL PROCESSING OF ENCODED BIT STRINGS(U) MARYLAND 1/1
UNIV COLLEGE PARK CENTER FOR AUTOMATION RESEARCH
A Y WU NOV 84 CAR-TR-98 AFOSR-TR-84-1181
UNCLASSIFIED F49620-83-C-0082 F/G 12/1 NL





AFOSR-TR- 84-1181

CAR-TR-98
CS-TR-1455

F49620-83-C-0082
November 1984

PARALLEL PROCESSING OF ENCODED BIT STRINGS

Angela Y. Wu

Dept. of Mathematics, Statistics,
and Computer Science
American University
Washington, DC 20016

HUMAN/COMPUTER INTERACTION LABORATORY

CENTER FOR AUTOMATION RESEARCH

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

CAR-TR-98
CS-TR-1455

F49620-83-C-0082
November 1984

PARALLEL PROCESSING OF ENCODED BIT STRINGS

Angela Y. Wu

Dept. of Mathematics, Statistics,
and Computer Science
American University
Washington, DC 20016

ABSTRACT

Many operations on strings of length n can be speeded up by a factor of p using p processors. String operations can also be speeded up, even when a single processor is used, by compactly encoding the strings, e.g. using run length code. This paper shows how to combine these two approaches by using p processors to process compactly encoded strings.

Research sponsored by the Air Force Office of Scientific Research (AFSC), under Contract F49620-83-C-0082. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The help of Janet Salzman in preparing this paper is gratefully acknowledged. The parallel merging algorithm is based in part on the work of Simon Kasif [7], and its application to Boolean operations was suggested by Azriel Rosenfeld.

1. Introduction

In the processing of a one-dimensional string of n symbols, the time complexity f(n) of a nontrivial sequential algorithm can at best be O(n) since the algorithm has to look at each symbol of the string at least once. One way to speed up the task is to use multiprocessor systems to process the string in parallel. In this case, the time complexity may become f(n)/p when p processors are used.

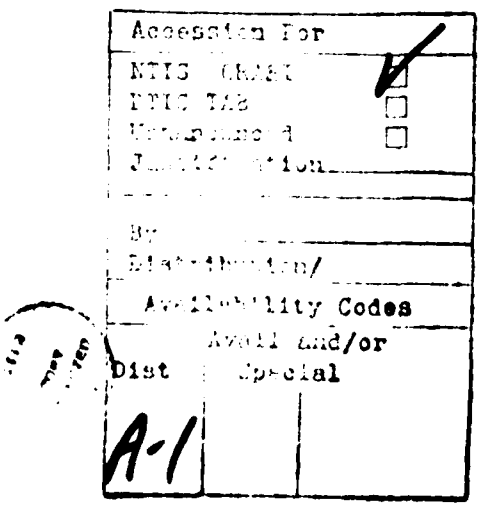
Another method to possibly speed up the task is to represent the string in some compact way instead of simply as a linear list of symbols. For example, a binary string of length n can be represented by its run length code, that is, by a string of m integers $a_1a_2a_3a_4\ldots a_m$ where $a_i>0$ except for a_1 which may be zero, a_{2i-1} specifies the number of 0's and a_{2i} specifies the number of 1's. For example, the run length code 0,3,2,4,1 represents 1110011110. In general m<n. These compact representations can often be processed quite efficiently by one processor.

A one-processor system may even be able to process a compact representation of a string faster than a multiprocessor system working with the original long string. To see this, consider the task of finding the number of 1's in the string. Using the run length code representation, a 1-processor system can accomplish this in $\frac{m}{2}$ steps by summing the a_{2i}'s $(1\leq i\leq\frac{m}{2})$. A p-processor system working with the original string of length n needs at least $\frac{n}{p}$ steps for each processor to find the number

of 1's it has, and it then takes log p steps to add the partial sums up. $\frac{n}{p} + \log p$ may be larger than $\frac{m}{2}$.

Using p processors to manipulate the compact representations in principle should achieve further speed up. However, the compactness of the representations often makes this difficult.

In this paper, we study various representations of bit strings and parallel algorithms to process these representations using a multiprocessor system. Section 2 describes the parallel processing model we use. Sections 3 and 4 discuss various compact representations of strings, and their conversions to each other. Section 5 presents algorithms to process run length coded strings. Section 6 briefly discusses the extension of this work to representations of two-dimensional objects.



2. Computational model

The parallel computational model used in this paper is a p-processor synchronous system. Each processor is identical, has its own local memory, and has a unique address. At each time step, a processor can send out one message and receive one message. Associated with each message is either a destination address or a pattern. A message sent by destination can only be accepted (picked up) by the processor with that address. A message sent by pattern can be accepted by any processor with that specified pattern. It is possible that a processor is the intended receiver of more than one message at a particular time step. In this case, only one of the messages (the first one that arrives) will be picked up, and the receiving processor does not know that there are other messages not picked up by it. At the end of a time step, a processor can tell whether its outgoing message was picked up by some processor(s), even though it cannot tell which processor(s) picked up the message if it was sent by pattern.

This model of communication is based on the ZMOB multiprocessor system which is being built at the University of Maryland. The processors are Z80 microprocessors and they are connected by a fast "conveyor belt" which consists of p shift registers. Currently, ZMOB is operational with p=32 and it is expected to reach p=256 when the system is complete. A detailed description of ZMOB can be found in [1-4]. The algorithms in this paper can be executed on ZMOB.

This ZMOB model of computation allows a flexible c nication network. Its address scheme permits one to re figure the network easily into any graph configuration. (It may take d time steps to simulate a node in the graph having d incoming arcs).

ZMOB is not so strong as shared memory models of parallel computation. In ZMOB, many processors can receive the same information from one processor; this is the same as reading from the same memory location. However, a processor can send out only one message at a time. Thus, it is not possible for two different processors to read from two locations within the same processor's local memory at the same time step. Restricted simultaneous writes (as long as the same value is written) can be performed as long as each processor's local memory is rewritten with only one value.

3. Bit string representations

This section discusses various ways to represent bit strings, and their suitability in a multiprocessor environment. On ways of representing binary arrays see [5].

(a) The bit string itself

A bit string $b_1b_2...b_n$ can be represented as a linear list of 0's and 1's -- for example, 001111100011. If we have p processors, we can partition the string into p parts of equal length (perhaps with the exception of the last part, which may be shorter) so that each processor is responsible for one segment of length $\frac{n}{p}$. It would be useful for each processor to know its segment number. This can be done implicitly by letting processor i contain the ith segment; or it can be done explicitly by storing the number i or the position (index, coordinate) of its first element.

(b) Run length code

In this representation, a bit string is specified by a sequence of integers $a_1a_2...a_m$ where $a_i>0$ for $i>1$ and $a_1\geq 0$. The bit string consists of a_1 consecutive 0's (a run of a_1 0's), followed by a_2 1's, followed by a_3 0's, followed by a_4 1's, etc. For example, 2542 represents the bit string 0011111000011 and 02542 represents 1100000111100. The run length code can be partitioned into p segments of $\lceil\frac{m}{p}\rceil$ (except for the last) runs each. Each processor will then be responsible for $\lceil\frac{m}{p}\rceil$ runs. The number of bits each processor represents may differ. It would be useful for each processor to know the coordinate (position in the bit string) of the first bit that it represents.

(c) Run-end code

This is similar to the run length code except that instead of specifying the lengths of all the runs, it specifies the positions of the beginning and the end of each run of 1's. In general, this uses more storage than specifying the lengths. But for a multiprocessor system, we are more concerned with speed since in general there is enough memory space to store the run ends. We can partition the run-end code in the same way as in the run length code case. The run-end code also allows the possibility of having the runs in any random order, as long as C_{2i-1}, C_{2i} specifies a run of 1's, and it is not necessary that $C_{2i-1} < C_{2i+1}$. However, to use the unsorted runs efficiently, we would often have to sort them first. Therefore, in this paper, we will only consider run ends in increasing order.

(d) Bin-tree

This is a one-dimensional equivalent of a quadtree representation for a two-dimensional array. More specifically, a bit string is represented by a binary tree as follows: The root of the binary tree represents the entire string. If the string that a node represents is not homogeneous (all 0's or all 1's) then the string is divided into two halves. The first half is then represented by the left child of the node and the second half by the right child. This division process stops when the string is homogeneous; in particular, it certainly stops when the string is of length 1.

The bin-tree takes more space to store than the run length or run end code. A run of 1's or a run of 0's can be distributed among several leaf nodes. In a multiprocessor system, if we use a processor to represent one node, then the number of runs we can represent using p processors is less than p. Clearly, we can do better using the run length or the run end code. Note that the bin-tree may be of value in sequential processing because it allows one to find a particular bit or block of bits in O(log n) time if the bit string is of length n.

(e) Run-binary-search tree

The runs of 1's in the string are stored in the nodes of a binary tree using coordinates as the key. By the same reasoning as in the bin-tree case, this is not particularly useful in a multiprocessor environment. In a single processor system, if the run ends are stored sequentially in increasing order, we already implicitly have a balanced binary tree. This representation is useful only if there are a lot of dynamic insertions and deletions.

Since the tree representations are not very useful, we will not consider them further.

4. Conversion between conversion

Since run length codes and run-end codes are very similar to each other, clearly they can be converted to one another in time proportional to the number of runs per processor. In this section we consider the conversion between run length code and the bit string. We will assume that processor i+1 is to the right of processor i, and will refer to it as the right hand neighbor of processor i.

4.1 Bit string to run length code

Suppose we are given a bit string of length n represented as a linear list of 0's and 1's and partitioned into p equal length parts, where each of p processors contains one segment of length $\frac{n}{p}$. It is obvious that in time $\frac{n}{p}$, each processor can convert its $\frac{n}{p}$ bits into run lengths. However, a processor needs to collate its first and last runs with its neighbors. A very long run of 0's and 1's may be distributed in k processors. After all the runs are obtained, they have to be redistributed so that each processor has the same number of runs.

After each processor converts its bit string into runs with the value 0 or 1 (for brevity, we will refer this as the color of the run), each processor sends (without retaining) its rightmost run (together with its color) to its right-hand neighbor if it has more than one run. Otherwise, only the color of the run is sent. Each processor receiving a value attaches it to or merges it with its first run depending on whether the two colors are the same. Each processor also now knows that

(1) It does not have part of a long run (a run which is now in more than one processor), if it has just accepted a run and sent out a run.

or (2) It has the beginning of a long run, if it initially had only one run and accepted a run from its neighbor.

or (3) It has the end of a long run, if its left neighbor is of length $\frac{n}{p}$, and its own first run has the same color and is of length $<\frac{n}{p}$.

or (4) Its left neighbor is the end of a long run, if its left neighbor's last run is of length $\frac{n}{p}$ and its own first run is of a different color.

or (5) It is the middle and possibly the end of a long run, if both its left neighbor and itself have only one run of length $\frac{n}{p}$ with the same color.

To clarify situation (5), the processor who knows its neighbor is the end of a long run sends a message to its left neighbor.

Now each processor i containing the beginning of a long run sends its address and run length to its right neighbor i+1. At the next step, both i and i+1 send the address i and run length to the right neighbors at distance 2 away, i.e., to i+2 and (i+1)+2. Only the middle and end processors who have not yet received any beginning address accept the address. At the next step processors i,i+1,i+2,i+3 send the address and run length to i+4,i+5,i+6,i+7. Continuing in this way, in less than or equal to O(log p) time, any processor that is the end of a long run has found the beginning address of its run and it can calculate the length of the run and store it. Any middle processor indicates it contains zero runs as soon as the beginning address reaches it.

Now all the runs are collated and each processor knows the number of runs it has (≥ 0). The system needs to distribute the runs evenly to the p processors.

We can calculate the total number of runs by simulating a binary tree structure, using processor $\frac{p}{2}$ as the root, processors $\frac{p}{4}$ and $\frac{3p}{4}$ as its left and right children, etc. Since p is known to all the processors, any processor knows if it is a leaf node (has depth p). At step 2i-1 ($i \geq 1$), a processor at depth i+1 accepts the number of runs its left child (at depth i) has. At step 2i, a processor at depth i+1 accepts the number of runs from its right child. The sum of the runs its two children have and it has is then sent up to the next level in the next two steps. In 2 log p steps, the root processor has the total number of runs (say m) in the bit string. Each processor also knows the number of runs in its left and right subtrees. The root can determine the number of runs each processor needs to have and broadcast this number to all the processors. In order for a processor to know to which processor its runs should be moved, it needs to have its runs numbered. Each processor finds the values Lcount and Rcount where Lcount(node) = number of runs before the first run in the node's left subtree and Rcount(node) = number of runs before the first run in the right subtree. Hence Lcount (root) = 0 and Rcount (root) = number of runs in its left subtree + number of runs in the root. The root sends it Lcount and Rcount to its

left and right children. Each child node can then set its Lcount = number just received from its parent, Rcount = number just received from its parent + number of runs in its left subtree + number of runs it itself contains. After 2 log p steps, each processor knows the numbers of its runs and thus the destinations of its runs. The distribution of the runs must be "orchestrated;" otherwise there may be many runs sent to the same processor simultaneously. At the ith distribution step, each processor sends out to the destination processor a run which is to be the ith run in that processor. In this case, at each step, at most one run is sent to each processor. If a processor contains more than one ith run, say k of them, then it must have contained at least $(k-1)\frac{m}{p}+1$ runs. The other ith runs are sent at steps $i+\frac{m}{p}$, $i+2\frac{m}{p}$, etc. Since each processor initially has $\frac{n}{p}$ bits, the maximum number of runs it has is $\frac{n}{p}$. Therefore, the distribution of runs takes time $\leq\frac{n}{p}$. In summary a bit string can be converted into evenly distributed run length code in $O(\frac{n}{p} + \log p)$ time.

4.2 Run length code to bit string

Suppose the run length code (having m runs) of a string is distributed in p processors with $\frac{m}{p}$ runs each. If no other information, such as the beginning coordinates, is given to the processors, then in $O(\frac{m}{p})$ time, each processor can find the total number of bits its runs represent by summing the run lengths. Simulating a binary tree as in Section 4.1 allows the root to find n, the total length of the entire string in O(log p) time. Using the method used in Section 4.1, in O(log p) time each processor knows the coordinates of its runs, and thus the destination of each of its runs. Note that some of the runs may have to be split among several processors. One way to distribute the runs is to simply cyclically shift each run to the right until it finally arrives its destination. This takes $O(\frac{m}{p} + p)$ time. After each processor receives its runs, it can convert them into bits.

5. Operations on run length coded strings

5.1 Operations involving only one string

In this section, we assume that bit strings of length n are represented by their run length codes. Each processor knows the coordinate of the first bit it represents. Each of the p processors has the same number ($=\frac{m}{p}$) of runs.

(a) Finding the total number of 1's in the string

Each processor finds the number of 1's it represents in $O(\frac{m}{p})$ steps by simply summing the lengths of the runs of 1's it contains. It takes O(log p) time steps to add up these p sums by implicitly simulating a binary tree as follows: At step 1, processor $2j-1$ $(j=1,2,\ldots,\frac{p}{2})$ sends its value to processor $2j$ which adds the value it receives to its own value. Each processor $2j$ $(1\leq j\leq\frac{p}{2})$ now has the number of 1's in processors $2j-1$ and $2j$. At step 2, processor 2^2j-2 $(j=1,2,\ldots,\frac{p}{4})$ sends its (new) value to processor 2^2j which adds the value it receives to its own value. At step i, processor 2^ij-2^{i-1} $(j=1,\ldots,\frac{p}{2^i})$ sends its value to processor 2^ij where an add is performed, unless 2^ij is larger than p. If $2^ij>p$ then 2^ij-2^{i-1} sends its value to processor p who adds the value it receives to its own value. At the end of k-1 $(2^{k-1}<p\leq 2^k)$ steps of sending and adding values, processor p has the total sum. If necessary, it can broadcast the result to all p processors.

(b) Finding local patterns in a string

A bit pattern is a sequence of 0's and 1's specified by its run length code. The pattern is local if it contains k runs and

$k \leq \frac{m}{p}$. Each processor can use the Knuth-Morris-Pratt algorithm [6] to find occurrences of the pattern in $O(\frac{m}{p}+k)$ time steps as long as we consider it a match for the first and last run of the pattern if the corresponding runs in the string are longer than the first and last runs in the pattern. If a processor (i) finds that the last few runs it contains match the beginning runs of the pattern, it sends this information to its right-hand neighbor (i+1) which checks if the pattern continues to match. Processor i+1 stops the pattern finding process after either this (across processor boundary) matching is successful or if a temporary failure causes the first run of the pattern no longer to be in processor i.

(c) Point in interval

Given a coordinate i, we want to find the value (0 or 1) of the ith bit in the bit string. When a processor receives the address i, it compares i with the address of its first bit. If i $\leq$ address (first bit) then '<' otherwise '>' is sent to its left-hand neighbor (except for processor 1). A processor that has result $\leq$ and that receives > from its right-hand neighbor (processor p has no right-hand neighbor and assumes it receives >) knows that it contains the ith bit. It then scans the run lengths in order and adds them up until it reaches a run which contains the ith bit; the value of this run is reported. This takes $O(\frac{m}{p})$ steps. If the runs in the processor are specified by run ends, then a binary search can be performed and the value of the ith bit can be found in $O(\log \frac{m}{p})$ time.

(d) Finding the address of the ith 1 in the string

Each processor can find the number of 1's it contains in $O(\frac{m}{p})$ time. Simulating a binary tree as in Section 4.1 allows processor j to know the total number of 1's that processor 1,2,...,j-1 has in O(log p) time. Hence the processor containing the ith 1 in the string can find the address in another $\frac{m}{p}$ steps.

(e) Finding the longest run of 1's

Each processor can find the length and address of its longest run. The length ℓ_i and address (i,a_i) $(i=2j-1)$ are then sent to i+1 where a comparison of ℓ_i and ℓ_{i+1} is made. The length and address of the longer run are sent from processors 2^2j-2 to 2^2j $(1\leq j\leq\frac{p}{4})$. Continuing this way, the length and address of the longest run of 1's will be in processor p after $O(\frac{m}{p} + \log p)$ time steps.

(f) Finding the centroid of the bit string

In $O(\frac{m}{p})$ time, each processor can find the total number of 1's it has and the sum of the coordinates of the 1's it contains. Then, as in (a) of this section, the total number of 1's in the string and the sum of the coordinates of the 1's in the entire string can be determined by processor p in O(log p) steps. Division gives the coordinate of the centroid of the string. This process takes $O(\frac{m}{p} + \log p)$ time.

5.2 Operations on two strings

Suppose the strings are represented by run length codes and each run's beginning and ending positions (coordinates) in the original bit string are given. These coordinates are a sorted list of numbers. Let $a_1, a_2, \ldots, a_s, a_{s+1}, \ldots, a_{2s}, a_{2s+1}, \ldots, a_{qs}$ $(q=\frac{p}{2})$ be the coordinates of the runs of one string, where s is even, $a_{(i-1)s+1}, \ldots, a_{is}$ are contained in processor i $(1 \le i \le q)$, and a_{2j-1}, a_{2j} are the beginning and ending coordinates of a run. Let $b_1, b_2, \ldots, b_t, b_{t+1}, \ldots, b_{2t}, b_{2t+1}, \ldots, b_{qt}$ $(q=\frac{p}{2})$ be the coordinates of the runs of the other string, where $b_{(i-1)t+1}, \ldots, b_{it}$ are in processor $\frac{p}{2}+i$ $(1 \le i \le q)$.

As indicated in [7], merging of these lists can be used to perform Boolean operations on the strings. We will first present an algorithm to merge two sorted strings of integers together. In the following, let PE i denote processor i.

Step 1 Processor i finds the index of the processor f_i $(q < f_i \le p)$ such that $b_{(f_i-1)t} < a_{is} \le b_{f_i t}$ (for $1 \le i \le q$) and $a_{(f_i-1)s} < b_{it} \le a_{f_i s}$ (for $q+1 \le i \le p$) using a divide and conquer method.

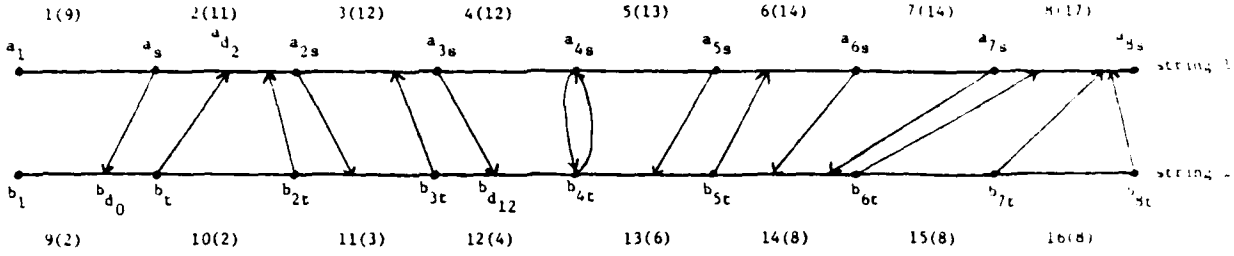
$PE_{\frac{q}{2}}$ broadcasts the value of $a_{\frac{q}{2}s}$ to PE's $q+1, \ldots, p$. Each PE j $(q < j \le p)$ compares b_{jt} with the value it receives and sends (if $j \ne p$) the result $< (b_{jt} < a_{\frac{q}{2}s})$ or $\ge (b_{jt} \ge a_{\frac{q}{2}s})$ to its right-hand neighbor PE $j+1$. If PE p (being a last processor) finds its result is $<$, then $b_{qt} < a_{\frac{q}{2}s} < a_{\frac{q}{2}s+1} < \ldots < a_{qs}$. PE p therefore sends the address $p+1$ to the PE's 1 through q, so that PE's $\frac{q}{2}+1, \ldots, q$ know that they are larger than all the elements in the second

string and PE's $1,\ldots,\frac{q}{2}-1$ know that they have to send their last elements to PE's $q+1,\ldots,p$. If PE $q+k$'s result is $\geq$ but PE $q+k-1$'s result is $<$, (PE $q+1$ being a beginning PE ignores results from PE q), then PE $k+p$ knows that $b_{(k-1)t}<a_{\frac{q}{2}s}\leq b_{kt}$ and sends its address to PE's $1,\ldots,q$. PE $\frac{q}{2}$ records the address it receives and stores it as $f_{\frac{q}{2}}$. PE $q+k-1$ marks itself as a new last processor and PE $q+k$ marks itself as a new first processor. Now PE's $1,2,\ldots,\frac{q}{2}-1$ have f values in $q+1,\ldots,q+k$, and PE's $\frac{q}{2}+1,\ldots,q$ have f values in $q+k,\ldots,p$. Thus each string is divided into two parts. Now simultaneously, PE $\frac{q}{4}$ and PE $\frac{3q}{4}$ can find $f_{\frac{q}{4}}$ and $f_{\frac{3q}{4}}$. Recursively in $O(\log_2 p)$ steps all the f_i's for $1\leq i<q$ are found. Similarly, the f_i's for $q+1<i\leq p$ can be found in $O(\log_2 p)$ steps.

Step 2 Every processor i (except q and p) sends its last value (a_{is} or b_{it}) to its right-hand neighbor processor i+1. Call this value C_{i+1}. Set $C_1=0$ and $C_{q+1}=0$.

Each processor sends out its list of values (including C_i) one at a time. Processor i picks up the values sent by PE f_i if $f_i\neq q+1$ or $p+1$. This uses the pattern matching communication method. Note that for each i, there is only one f_i but several processors may have the same f values. Processor i merges and keeps the values it originally contained and it receives, which are larger than both C_i and C_{f_i} and $\leq C_{i+1}$ and C_{f_i+1}. Clearly, this can be accomplished in $O(s+t)$ time.

Since the values in the processors are in increasing order, each value in the original lists appears in the final list only once with the exception that if $a_{is}=b_{f_i t}$ or $a_{f_i s}=b_{it}$, then the values in processors i and f_i are identical as the following figure shows:



The f_i's are in parentheses. Processor 1 contains the values in $a_1,\ldots,a_s$ and $b_1,\ldots,b_{d_9}$. Processor 9 contains the values in $a_{s+1},\ldots,a_{d_2}$ and $b_{d_9+1},\ldots,b_t$, etc. Processors 4 and 12 both contain values in $a_{3s+1},\ldots,a_{4s}$ and $b_{d_{12}+1},\ldots,b_{4t}$. Since processor q+i knows $b_{it}=a_{f_{q+i}s}$, it can simply indicate it will contain no values and ignore the merging process and leave it for processor f_{q+i} to do the merging.

Step 3 Processor i knows that the values it contains belong to segment $i+f_i-q-1$ of the final merged list. This is true because of the way the values are obtained. Therefore, if we want to have the values in consecutive processors in order, we can have each processor i send out its values one by one to processor $i+f_i-q-1$. This can be accomplished in O(s+t) time.

This algorithm shows that two sorted strings of length m_1, m_2, each evenly distributed in $q=\frac{p}{2}$ processors, can be merged in $O(\frac{m_1}{q}+\frac{m_2}{q}+\log q)$ time.

If we want to find the AND of two bit strings, we just need to make sure that the beginning and ending coordinates of a run get sent to the processors whenever one of them is sent in step 2. Instead of merging, the AND operation is done. If we want to find the OR of two bit strings, we simply do the OR operation instead of the AND. We must also check for possible collating of runs after the AND or OR is done, as in Section 3.

6. Concluding remarks

We have shown that many operations on run length representations of bit strings can be speeded up using a multiprocessor system. In most cases, an overhead of log p is needed for an orderly communication scheme to accumulate information from the p processors.

A string can be regarded as a 1-dimensional array; similarly, a digital picture is a 2-dimensional array. There are various compact representations of binary pictures, including run length code (row by row), chain code (of the borders between regions of 0's and regions of 1's), quadtrees, or the medial axis transform [5]. It is of interest to see how we can use a multiprocessor system to perform operations using these compact representations.

It is not clear how one should distribute the representation to the p processors. It was shown in [4] that the best way to distribute an n×n picture specified by its pixels' gray levels is to partition the picture into p subpictures of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ each. Then each processor can be responsible for one subpicture.

If a picture is represented by the run length codes of its rows, operations such as finding the number of black pixels in a picture, or taking the AND or OR of two pictures, can be done row by row using the algorithms described in this paper. This is possible because the two-dimensional properties of pictures are not used in these operations. An operation such as finding the centroid of a two-dimensional picture requires knowledge of the two-dimensional coordinates of the pixels; however, each pixel does not interact

with the other pixels. A slight modification of the algorithm in this paper solves the problem.

Further study is needed to develop algorithms for truly two-dimensional operations or to handle representations other than run length codes.

References

1. C. Rieger, ZMOB: handware from a user's viewpoint, Proc. IEEE Workshop Pattern Recognition and Image Processing (PRIP), Dallas, TX, August 1981.

2. C. Rieger, ZMOB: Doing it in parallel!, Proc. IEEE Workshop CAPAIDM, Hot Springs, VA, November 1981.

3. C. Rieger, R. Trigg and R. Bane, ZMOB: A new computing engine for AI, Proc. IJCAI-81, Vancouver, B.C., Canada, August 1981.

4. T. Kushner, A. Wu, and A. Rosenfeld, Image processing on ZMOB, IEEE Trans.Computers, vol. C-31, 943-951 (1982).

5. A. Rosenfeld and A. Kak, Digital Picture Processing, Academic Press, New York (second edition, vol. 2), chapter 11 (1982).

6. D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, SIAM J. Computing, vol. 6, 323-350 (1977).

7. S. Kasif, Parallel searching and merging on ZMOB, University of Maryland Computer Science TR-1405, June 1984.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CAR-TR-98; CS-TR-1455.	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 85-0016
6a. NAME OF PERFORMING ORGANIZATION University of Maryland 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code) Center for Automation Research College Park MD 20742	7b. ADDRESS (City, State and ZIP Code) Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332-6448
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR 8b. OFFICE SYMBOL (If applicable) NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-83-C-0082
8c. ADDRESS (City, State and ZIP Code) Bolling AFB DC 20332-6448	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 61102F; PROJECT NO.: 2304; TASK NO.: A7; WORK UNIT NO.:
11. TITLE (Include Security Classification) PARALLEL PROCESSING OF ENCODED BIT STRINGS	
12. PERSONAL AUTHOR(S) Angela Y. Wu	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) NOV 84 15. PAGE COUNT 24
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Parallel processing; bit strings; run length codes.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Many operations on strings of length n can be speeded up by a factor of p using p processors. String operations can also be speeded up, even when a single processor is used, by compactly encoding the strings, e.g., using run length code. This paper shows how to combine these two approaches by using p processors to process compactly encoded strings.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Robert N. Buchal	22b TELEPHONE NUMBER (Include Area Code) (202) 767- 4939 22c OFFICE SYMBOL NM

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

END

FILMED

3-85

DTIC